OOPS [Important Only]

1. Basics of OOPs

Concept:

Object-Oriented Programming (OOP) is a programming paradigm that uses objects and classes to design and develop applications. It focuses on four principles:

- **Encapsulation:** Binding data and methods together.
- **Inheritance:** Reusing properties and methods of existing classes.
- **Polymorphism:** Using one entity in multiple forms.
- **Abstraction:** Hiding implementation details from the user.

Example:

Imagine a `Car` class with attributes like `brand` and methods like `drive()`. You can create different objects of this class for different cars.

---

2. Fundamentals of Class & Object

Concept:

- A **class** is a blueprint for creating objects.
- An **object** is an instance of a class containing data and behavior.

Code Example:

```
class Car {
    String brand;
    int speed;

    void drive() {
        System.out.println(brand + " is driving at " + speed + " km/hr");
    }
}

public class Main {
```

```
    public static void main(String[] args) {
        Car car = new Car();
        car.brand = "Tesla";
        car.speed = 100;
        car.drive();
    }
}
```

---

3. `new` Keyword

Concept:

The `new` keyword is used to create objects by allocating memory dynamically.

Code Example:

```
Car car = new Car();
```

---

4. Reference Variables

Concept:

A reference variable stores the memory address of the object it refers to.

Code Example:

```
Car car1 = new Car();
Car car2 = car1; // Both reference the same object
```

---

5. Member Methods of a Class

Concept:

These are methods defined in a class to perform operations on its data.

Code Example:

```
class Calculator {
    int add(int a, int b) {
        return a + b;
    }
}
```

6. Constructors

Concept:

A constructor is a special method used to initialize objects. It has the same name as the class and no return type.

Code Example:

```
class Car {
    String brand;

    Car(String brand) {
        this.brand = brand;
    }
}
```

7. `finalize()` Method

Concept:

The `finalize()` method is called by the garbage collector before an object is destroyed.

Code Example:

```
java
Copy code
@Override
protected void finalize() {
```

```
    System.out.println("Object is being destroyed.");
}
```

---

8. Overloading Member Methods

Concept:

Having multiple methods with the same name but different parameters.

Code Example:

```
class MathUtils {
    int add(int a, int b) {
        return a + b;
    }

    double add(double a, double b) {
        return a + b;
    }
}
```

---

9. Overloading Constructors

Concept:

A class can have multiple constructors with different parameter lists.

Code Example:

```
class Car {
    String brand;
    int speed;

    Car(String brand) {
        this.brand = brand;
    }
```

```
    Car(String brand, int speed) {
        this.brand = brand;
        this.speed = speed;
    }
}
```

---

10. Passing and Returning Objects with Methods

Concept:

Methods can accept and return objects.

Code Example:

```
class Car {
    String brand;

    Car(String brand) {
        this.brand = brand;
    }

    Car updateBrand(String newBrand) {
        return new Car(newBrand);
    }
}
```

---

11. Access Control

Concept:

Access control modifiers define the scope of classes, variables, and methods:

- `public`: Accessible everywhere.
- `protected`: Accessible within package and subclasses.
- `default`: Accessible within the package.
- `private`: Accessible within the class.

Code Example:

```
class Example {
    private int data; // Private access
}
```

---

12. Static Methods

Concept:

A `static` method belongs to the class, not to an instance.

Code Example:

```
class MathUtils {
    static int square(int x) {
        return x * x;
    }
}
```

---

13. Static Variables

Concept:

A `static` variable is shared among all instances of a class.

Code Example:

```
class Counter {
    static int count = 0;
}
```

---

14. Static Block

Concept:

A `static` block is executed when the class is loaded.

Code Example:

```
class Example {
    static {
        System.out.println("Static block executed.");
    }
}
```

---

15. Using `final` Keyword

Concept:

- `final` variable: Cannot be reassigned.
- `final` method: Cannot be overridden.
- `final` class: Cannot be inherited.

Code Example:

```
final int MAX_SPEED = 120;
```

Inheritance in Java

Inheritance is a mechanism where one class (child/subclass) can inherit properties and behaviors (fields and methods) from another class (parent/superclass). It promotes code reuse and establishes a hierarchical relationship among classes.

---

1. Basics of Inheritance

Concept: Inheritance allows a subclass to acquire the properties and methods of a superclass.

Example:

```
class Animal {
    void eat() {
        System.out.println("This animal eats food.");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("The dog barks.");
    }
}

public class BasicInheritance {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.eat();  // Accessing superclass method
        dog.bark(); // Accessing subclass method
    }
}
```

Explanation:

- `Dog` inherits the `eat` method from `Animal`.
- The `Dog` class can also have its own method (`bark`).

---

2. Members Accessibility in Inheritance

Concept: Members of the superclass are accessed in the subclass based on their visibility (public, protected, private).

Example:

```
class Parent {
    public int publicVar = 1;
    protected int protectedVar = 2;
```

```
    private int privateVar = 3;

    void displayParent() {
        System.out.println("Public: " + publicVar);
        System.out.println("Protected: " + protectedVar);
        // Private is accessible within the same class only
    }
}

class Child extends Parent {
    void displayChild() {
        System.out.println("Accessing public: " + publicVar);
        System.out.println("Accessing protected: " + protectedVar);
        // System.out.println(privateVar); // Not accessible
    }
}
```

---

3. Using `super` Keyword

Concept: The `super` keyword is used to access superclass methods or constructors.

Example:

```
class Vehicle {
    String type = "Vehicle";

    void display() {
        System.out.println("This is a vehicle.");
    }
}

class Car extends Vehicle {
    String type = "Car";

    void display() {
        super.display(); // Calls superclass method
```

```
        System.out.println("This is a car.");
    }

    void showType() {
        System.out.println("Super type: " + super.type); // Accessing supercla
ss field
        System.out.println("Current type: " + this.type);
    }
}

public class SuperExample {
    public static void main(String[] args) {
        Car car = new Car();
        car.display();
        car.showType();
    }
}
```

4. Multilevel Inheritance

Concept: A class can inherit from another class, which in turn inherits from another class.

Example:

```
class Animal {
    void eat() {
        System.out.println("This animal eats food.");
    }
}

class Mammal extends Animal {
    void breathe() {
        System.out.println("Mammals breathe air.");
    }
}
```

```
class Dog extends Mammal {
    void bark() {
        System.out.println("Dogs bark.");
    }
}

public class MultilevelInheritance {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.eat();
        dog.breathe();
        dog.bark();
    }
}
```

5. Sequence of Execution of Constructors

Concept: Constructors are called in the order of inheritance from the topmost parent class to the child class.

Example:

```
class Parent {
    Parent() {
        System.out.println("Parent Constructor");
    }
}

class Child extends Parent {
    Child() {
        System.out.println("Child Constructor");
    }
}

class GrandChild extends Child {
    GrandChild() {
        System.out.println("GrandChild Constructor");
```

```
    }
}

public class ConstructorExecution {
    public static void main(String[] args) {
        GrandChild obj = new GrandChild();
    }
}
```

Output:

```
Copy code
Parent Constructor
Child Constructor
GrandChild Constructor
```

---

6. Method Overriding

Concept: A subclass provides its own implementation for a method defined in the superclass.

Example:

```
class Parent {
    void display() {
        System.out.println("Display from Parent");
    }
}

class Child extends Parent {
    @Override
    void display() {
        System.out.println("Display from Child");
    }
}
```

```
public class MethodOverriding {
    public static void main(String[] args) {
        Parent obj = new Child();
        obj.display(); // Calls the Child's method
    }
}
```

7. Dynamic Method Dispatch

Concept: The method to be called is determined at runtime based on the object reference.

Example:

```
class Parent {
    void display() {
        System.out.println("Parent display");
    }
}

class Child extends Parent {
    void display() {
        System.out.println("Child display");
    }
}

public class DynamicDispatch {
    public static void main(String[] args) {
        Parent obj = new Child();
        obj.display(); // Runtime decides to call Child's display
    }
}
```

8. Abstract Classes

Concept: An abstract class cannot be instantiated but can have abstract and concrete methods.

Example:

```
abstract class Shape {
    abstract void draw(); // Abstract method

    void display() { // Concrete method
        System.out.println("Displaying shape.");
    }
}

class Circle extends Shape {
    @Override
    void draw() {
        System.out.println("Drawing a circle.");
    }
}

public class AbstractExample {
    public static void main(String[] args) {
        Shape shape = new Circle();
        shape.draw();
        shape.display();
    }
}
```

---

9. Preventing Overriding

Concept: Use the `final` keyword to prevent a method from being overridden.

Example:

```
class Parent {
    final void display() {
        System.out.println("This cannot be overridden.");
```

```
    }
}

class Child extends Parent {
    // void display() {} // Compilation error
}
```

10. Preventing Inheritance

Concept: Use the `final` keyword with a class to prevent it from being inherited.

Example:

```
final class Parent {
    void display() {
        System.out.println("This class cannot be inherited.");
    }
}

// class Child extends Parent {} // Compilation error
```

Key Takeaways

- Inheritance enables code reuse and improves maintainability.
- `super` keyword is essential to access parent class members.
- Method overriding and dynamic method dispatch are critical for runtime polymorphism.
- Abstract classes define a blueprint for subclasses.
- `final` keyword helps enforce restrictions on inheritance and overriding.

Polymorphism

Concept:

Polymorphism in Java is the ability of an object to take many forms. It allows methods to perform different tasks based on the object or data they operate on. Polymorphism promotes flexibility and reusability in code.

- **Purpose**: It simplifies code and enables dynamic behavior in programs.
- **Where It Is Used**:
  - Implementing dynamic method dispatch (runtime behavior)
  - Enhancing code readability
  - Achieving abstraction and encapsulation

Polymorphism in Java

Polymorphism is broadly categorized into:

1. **Compile-time Polymorphism (Static Polymorphism)**: Achieved using method overloading.
2. **Run-time Polymorphism (Dynamic Polymorphism)**: Achieved using method overriding.

---

1. Types of Polymorphism

Compile-Time Polymorphism

Compile-time polymorphism occurs when method calls are resolved at compile time.

Example (Method Overloading):

```
class Calculator {
    // Method with two integer parameters
    int add(int a, int b) {
        return a + b;
    }

    // Method with three integer parameters
    int add(int a, int b, int c) {
        return a + b + c;
    }
```

```
    // Method with double parameters
    double add(double a, double b) {
        return a + b;
    }
}

public class CompileTimePolymorphism {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println("Sum of 2 integers: " + calc.add(5, 10));
        System.out.println("Sum of 3 integers: " + calc.add(5, 10, 15));
        System.out.println("Sum of 2 doubles: " + calc.add(5.5, 10.5));
    }
}
```

Explanation:

- Method `add` is overloaded with different parameter lists.
- The compiler determines which version of `add` to call based on the arguments.

Run-Time Polymorphism

Run-time polymorphism occurs when method calls are resolved during runtime. It is achieved through **method overriding**.

Example (Method Overriding):

```
class Animal {
    void sound() {
        System.out.println("Animals make different sounds.");
    }
}

class Dog extends Animal {
    @Override
    void sound() {
```

```
        System.out.println("The dog barks.");
    }
}

class Cat extends Animal {
    @Override
    void sound() {
        System.out.println("The cat meows.");
    }
}

public class RunTimePolymorphism {
    public static void main(String[] args) {
        Animal myAnimal;  // Reference of parent class
        myAnimal = new Dog();
        myAnimal.sound(); // Calls Dog's sound()

        myAnimal = new Cat();
        myAnimal.sound(); // Calls Cat's sound()
    }
}
```

Explanation:

- `Animal` reference is used to call overridden methods in `Dog` and `Cat`.
- The method to be executed is determined during runtime.

---

2. Static and Dynamic Binding

- **Static Binding**: Method calls are resolved at compile time (e.g., method overloading).
- **Dynamic Binding**: Method calls are resolved at runtime (e.g., method overriding).

Example of Binding:

```
class BindingExample {
```

```
    void staticBinding() {
        System.out.println("This is static binding.");
    }

    void dynamicBinding() {
        System.out.println("This is dynamic binding.");
    }
}

public class Binding {
    public static void main(String[] args) {
        BindingExample obj = new BindingExample();
        obj.staticBinding(); // Compile-time resolution

        BindingExample dynamic = new BindingExample() {
            @Override
            void dynamicBinding() {
                System.out.println("Overridden dynamic binding.");
            }
        };
        dynamic.dynamicBinding(); // Run-time resolution
    }
}
```

3. Method Overloading

Concept:

Method overloading allows a class to have multiple methods with the same name but different parameters.

Real-Time Example:

```
class Payment {
    // Pay using card
    void pay(String cardNumber, double amount) {
        System.out.println("Payment of " + amount + " using card: " + cardNu
```

```
mber);
    }

    // Pay using UPI
    void pay(String upiId) {
        System.out.println("Payment using UPI: " + upiId);
    }

    // Pay with cash
    void pay(double amount) {
        System.out.println("Payment of " + amount + " using cash.");
    }
}

public class OverloadingExample {
    public static void main(String[] args) {
        Payment payment = new Payment();
        payment.pay("1234-5678-9012-3456", 1000);
        payment.pay("user@upi");
        payment.pay(500);
    }
}
```

4. Method Overriding

Concept:

When a subclass redefines a method of its superclass with the same signature.

Real-Time Example:

```
class Printer {
    void print() {
        System.out.println("Printing a document.");
    }
}
```

```
class ColorPrinter extends Printer {
    @Override
    void print() {
        System.out.println("Printing a color document.");
    }
}

public class OverridingExample {
    public static void main(String[] args) {
        Printer printer = new ColorPrinter(); // Upcasting
        printer.print(); // Calls overridden method
    }
}
```

5. Dynamic Method Dispatch

Concept:

Dynamic method dispatch is a mechanism where the method to be called is determined at runtime based on the actual object being referred to by a parent reference.

Real-Time Example:

```
class Shape {
    void draw() {
        System.out.println("Drawing a shape.");
    }
}

class Circle extends Shape {
    @Override
    void draw() {
        System.out.println("Drawing a circle.");
    }
}
```

```
class Rectangle extends Shape {
    @Override
    void draw() {
        System.out.println("Drawing a rectangle.");
    }
}

public class DynamicDispatchExample {
    public static void main(String[] args) {
        Shape shape;

        shape = new Circle();
        shape.draw(); // Runtime decides to call Circle's draw()

        shape = new Rectangle();
        shape.draw(); // Runtime decides to call Rectangle's draw()
    }
}
```

Key Takeaways

- **Polymorphism** enhances code flexibility and reusability.
- **Method Overloading** enables multiple methods with the same name but different parameters (compile-time).
- **Method Overriding** provides specific behavior in a subclass (runtime).
- **Dynamic Method Dispatch** ensures runtime decisions for method calls.
- **Static Binding** is associated with method overloading, while **Dynamic Binding** is tied to overriding.

Abstraction in Java

Concept

Abstraction is the process of hiding the implementation details and showing only the essential features of an object. It helps reduce complexity by providing

a simplified interface to interact with the object.

In Java, abstraction can be achieved in two ways:

1. **Abstract Classes**: Partially abstract classes with both implemented and unimplemented methods.
2. **Interfaces**: Fully abstract structures that define a contract for classes to implement.

Real-World Example

A **vehicle** can be abstracted as:

- All vehicles can **start** and **stop** (common methods).
- How they start (e.g., key ignition, push-button) is vehicle-specific (implementation hidden).

Abstract Class in Java

Concept

An abstract class is a class that is declared with the `abstract` keyword. It can have both:

- **Abstract methods**: Methods without implementation (declared with `abstract` keyword).
- **Concrete methods**: Methods with implementation.

An abstract class cannot be instantiated but can be extended by a subclass.

Code Example

```
// Abstract Class
abstract class Vehicle {
    abstract void start(); // Abstract method (no body)

    void stop() { // Concrete method
        System.out.println("Vehicle stopped.");
    }
}
```

```
// Subclass 1
class Car extends Vehicle {
    @Override
    void start() {
        System.out.println("Car starts with a key.");
    }
}

// Subclass 2
class Bike extends Vehicle {
    @Override
    void start() {
        System.out.println("Bike starts with a button.");
    }
}

public class Main {
    public static void main(String[] args) {
        Vehicle car = new Car();
        car.start(); // Calls overridden method in Car
        car.stop();  // Calls concrete method from Vehicle

        Vehicle bike = new Bike();
        bike.start(); // Calls overridden method in Bike
        bike.stop();  // Calls concrete method from Vehicle
    }
}
```

Explanation

1. `Vehicle` is an abstract class with an abstract method `start()` and a concrete method `stop()`.
2. Subclasses `Car` and `Bike` provide their own implementation for the `start()` method.
3. The abstract class allows partial abstraction by defining both common functionality (`stop()`) and specific behaviors (`start()`).

Abstract Method in Java

Concept

An abstract method is a method declared without a body (implementation) in an abstract class. It must be implemented by subclasses.

Code Example

```
abstract class Shape {
    abstract void draw(); // Abstract method
}

class Circle extends Shape {
    @Override
    void draw() {
        System.out.println("Drawing a circle.");
    }
}

class Rectangle extends Shape {
    @Override
    void draw() {
        System.out.println("Drawing a rectangle.");
    }
}

public class Main {
    public static void main(String[] args) {
        Shape circle = new Circle();
        circle.draw();

        Shape rectangle = new Rectangle();
        rectangle.draw();
    }
}
```

Key Takeaways

- Abstract methods define a "contract" for subclasses.
- Subclasses must implement abstract methods, ensuring consistent behavior.

Interface in Java

Concept

An interface is a reference type in Java that is similar to a class but only contains abstract methods (until Java 7). From Java 8 onwards, interfaces can have:

1. **Default methods** (with body).
2. **Static methods** (with body).

Interfaces are used to achieve 100% abstraction and support multiple inheritance.

Code Example

```
interface Animal {
    void sound(); // Abstract method

    default void sleep() { // Default method
        System.out.println("Animal is sleeping.");
    }
}

class Dog implements Animal {
    @Override
    public void sound() {
        System.out.println("Dog barks.");
    }
}

class Cat implements Animal {
    @Override
```

```
    public void sound() {
        System.out.println("Cat meows.");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal dog = new Dog();
        dog.sound(); // Calls overridden method
        dog.sleep(); // Calls default method

        Animal cat = new Cat();
        cat.sound();
        cat.sleep();
    }
}
```

Explanation

1. **Abstract methods**: `sound()` must be implemented by all classes that implement `Animal`.
2. **Default methods**: Provide a default implementation, reducing the need for duplicate code.

Nested Interface in Java

Concept

A nested interface is an interface defined within another interface or class. It is used when the nested interface is strongly associated with the enclosing class or interface.

Rules for Nested Interfaces

1. If defined inside a class, the nested interface must be declared `public`.
2. If defined inside an interface, it is implicitly `public` and `static`.

Code Example

```
class Machine {
    interface Operation { // Nested interface inside a class
        void start();
        void stop();
    }
}

class WashingMachine implements Machine.Operation {
    @Override
    public void start() {
        System.out.println("Washing Machine is starting.");
    }

    @Override
    public void stop() {
        System.out.println("Washing Machine is stopping.");
    }
}

public class Main {
    public static void main(String[] args) {
        Machine.Operation wm = new WashingMachine();
        wm.start();
        wm.stop();
    }
}
```